ImageEye: Batch Image Processing using Program Synthesis

CELESTE BARNABY, University of Texas at Austin, USA
QIAOCHU CHEN, University of Texas at Austin, USA
ROOPSHA SAMANTA, Purdue University, USA
IŞIL DILLIG, University of Texas at Austin, USA

This paper presents a new synthesis-based approach for batch image processing. Unlike existing tools that can only apply global edits to the entire image, our method can apply fine-grained edits to individual objects within the image. For example, our method can selectively blur or crop specific objects that have a certain property. To facilitate such fine-grained image editing tasks, we propose a neuro-symbolic domain-specific language (DSL) that combines pre-trained neural networks for image classification with other language constructs that enable symbolic reasoning. Our method can automatically learn programs in this DSL from user demonstrations by utilizing a novel synthesis algorithm. We have implemented the proposed technique in a tool called ImageEye and evaluated it on 50 image editing tasks. Our evaluation shows that ImageEye is able to automate 96% of these tasks.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Program Synthesis, Neuro-symbolic Synthesis, Computer Vision

ACM Reference Format:
Celeste Barnaby, Qiaochu Chen, Roopsha Samanta, and Işıl Dillig. 2023. ImageEye: Batch Image Processing using Program Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 134 (June 2023), 26 pages. https://doi.org/10.1145/3591248

1 INTRODUCTION

Because many real-world scenarios require editing a very large number of images, existing photo editing software provides some support for image processing in batch mode. For example, popular software like Adobe Photoshop and Luminar allow users to process multiple files at the same time by specifying actions like resizing or converting to a specified file type.

Despite the popularity of such tools, batch editing capabilities of existing software are extremely limited and can only perform edits globally to the *entire* image. However, many image editing tasks of interest require fine-grained edits to specific parts of the image. For example, consider a scenario where someone wants to upload a collection of their photos after concealing the identities of certain people. Such a task requires performing selective edits (e.g., blurring) to certain parts of the image but not others. As another example, consider a batch processing task where someone wishes to adjust the white balance of certain types of objects, such as human faces. Because this requires combining programmatic edits with object classification, existing solutions fall short in successfully automating such image manipulation tasks.

134

Authors' addresses: Celeste Barnaby, University of Texas at Austin, USA, celestebarnaby@utexas.edu; Qiaochu Chen, University of Texas at Austin, USA, qchen@cs.utexas.edu; Roopsha Samanta, Purdue University, USA, roopsha@purdue.edu; Işıl Dillig, University of Texas at Austin, USA, isil@cs.utexas.edu.

2475-1421/2023/6-ART134
https://doi.org/10.1145/3591248

In this paper, we propose a new technique, based on program synthesis, for automating selective image editing tasks. Given a small set of user demonstrations (performed through a graphical user interface), our approach automatically synthesizes a program that can be applied to a much larger set of images. Because these programs are expressed in a neuro-symbolic domain-specific language (DSL) comprised of both logical operators and (pre-trained) neural networks, they can be used to perform *fine-grained edits* where different actions can be selectively applied to different parts of the image. As a result, our approach can automate image processing tasks that are well beyond the scope of existing tools.

At a high level, programs in our image editing DSL specify what actions (blur, crop etc.) to apply to what parts of the image. Thus, an image editing program can be viewed as a set of *extractor* and *action* pairs where each extractor selects a *part* of the image and the action specifies what operation to apply to that part. Because these extractors are expressed in a rich vocabulary involving both neural primitives and functional operators, our DSL makes it possible to combine object classification with relational reasoning.

Beyond proposing a DSL for batch image processing, a key contribution of this paper is a new program synthesis technique for learning programs in this DSL. At a high level, our approach reduces this problem to a more standard programming-by-example (PBE) task by utilizing the concept of *symbolic images*: rather than representing an image as a set of low-level pixels, we represent images as a mapping from object identifiers to their symbolic properties. This representation is obtained by applying segmentation to the input image and utilizing neural object classifiers to extract attributes of each detected object. Overall, this symbolic representation is crucial to our technique in two ways: First, it allows defining a formal semantics of our DSL in terms of sets of high-level objects as opposed to a 2D array of low-level pixels. Second, due to this symbolic representation, the learning task can be reduced to the problem of synthesizing an extractor function that produces a target set of objects from among all objects in the input image.

To solve the PBE problem in this context, our approach utilizes top-down enumerative search, as done in prior work [Feser et al. 2015; Le and Gulwani 2014; Smith and Albarghouthi 2019; Wang et al. 2017]. As standard, the idea is to maintain a worklist of *partial programs* (i.e., programs with unknown parts) that are gradually refined into a concrete implementation. However, basic enumerative search does not scale to the image manipuation tasks of interest in this work because images often contain *many* objects with *many* different attributes. As a result, the search space becomes enormous, necessitating novel pruning techniques that can be utilized to rule out redundant or infeasible partial programs. Specifically, our underlying PBE algorithm addresses scalability challenges of this domain through two key insights:

(1) **Equivalence reduction with term rewriting and partial evaluation:** Many partial programs enumerated during top-down search are bound to produce the same output image *no matter* how the unknown parts are instantiated. In other words, the basic search procedure ends up enumerating many *redundant* partial programs that can be safely thrown away. To detect such redundancies and prune the search space, our method leverages a combination of *term rewriting* and *partial evaluation* to reason about observational equivalence in the context of images. While prior work on program synthesis has used term rewriting and partial evaluation *in isolation*, we show that term rewriting is considerably more effective in this context when it is combined with partial evaluation.
(2) **Abstract semantics for images:** Some partial programs enumerated during search can *never* produce the target output image no matter how they are refined into a concrete implementation. To avoid such dead-ends in the search space, our method utilizes a novel abstraction (and its corresponding abstract semantics) for image editing programs. In particular, reasoning

backwards from the target image, our method infers the set of objects that *must* and *may* appear in an (unknown) subprogram and uses this information to identify infeasible partial programs in our image editing DSL.

We have implemented our proposed approach in a tool called ImageEye and evaluated it on 50 image processing tasks inspired by practical tasks and on-line forum discussions. Our evaluation shows that ImageEye can successfully automate 48 of these tasks and that it can infer the intended program after a small number of user demonstrations. We also perform comparisons against simpler synthesis baselines and present ablation studies to demonstrate the usefulness of our proposed synthesis technique.

To summarize, this paper makes the following key contributions:

- We describe the first solution for automating fine-grained image editing tasks.
- We present an image processing DSL that combines neural computer vision primitives with programmatic constructs for relational reasoning, and we define the formal semantics of this DSL in terms of the concept of *symbolic images*.
- We propose a novel synthesis algorithm for generating programs in our DSL from a set of user demonstrations. Our technique decomposes the overall synthesis problem into a set of independent PBE tasks and utilizes two key ideas (namely, abstraction-guided reasoning about images and combined use of partial evaluation and term rewriting) to allow this approach to scale to realistic image batches.
- We implement our approach in a new tool called ImageEye and evaluate it experimentally on dozens of image editing tasks involving a diverse set of images.

2 OVERVIEW

Usage scenario. Suppose that a user has a batch of several hundred images from a school recital. The user would like to identify all images that feature their daughter playing the violin and crop everything else out of those images. Our proposed tool, ImageEye, is useful for these types of tasks that are easy for a small number of images but grueling for a large batch.

To automate this task, the user loads their images into ImageEye and identifies a few images that feature their daughter playing a violin. For each image, they use the ImageEye graphical user interface to select their daughter's face and violin, and choose the "Crop" action to crop the the selected region. Under the hood, the ImageEye GUI uses computer vision models (for image segmentation and object recognition) to highlight the detected objects and allows the user to select each object individually. If the relevant objects are not detected in the current image, the user will quickly realize that this particular image is not useful for demonstrating their intent and move on to a different image.

Once the user has edited a few representative images, they press the "Synthesize" button, which invokes ImageEye's synthesis engine and searches for a program P that matches the user's demonstrations. ImageEye then applies P to all images in the batch, and produces a new set of edited images. Next, the user inspects the resulting images to decide whether the synthesized program is correct. If P fails to produce the intended edit for many images in the batch, then the synthesis result is likely incorrect and the user may provide one or more additional demonstrations as training examples. On the other hand, if the output images look as intended, the synthesized program likely captures the user's intent. However, there may be some imperfections due to shortcomings of the neural models used in the synthesized program. In this case, the user needs to manually edit a small number of images where the resulting image differs from the expected output, but this is still much more convenient than manually editing *all* images in the batch.

Shortcomings of existing techniques. While existing image editing tools like Photoshop and GIMP allow some forms of batch processing, they only support simple manipulations (like resizing or applying a filter) to the *entire* image. Notably, such tools do not allow batch processing tasks that differ based on the *content* of each image. Because our motivating example requires reasoning about the presence of specific objects in each image and applying a cropping action accordingly, such tools are not useful for automating this task.

On the other hand, tools like Amazon Rekognition use pre-trained neural models for object detection. Given an image, these tools can identify and locate a wide range of objects, including text and human faces. In addition, they can recognize the same face across different images and discern several interesting properties of human faces, such as their approximate age and whether the person is smiling. However, they neither provide functionality for editing individual images nor for batch processing. Hence, such computer vision models are also not directly useful for performing the task in our motivating example.

Our approach. In contrast to existing techniques, our approach combines the relative strengths of programmatic batch image processing with computer vision techniques (image segmentation and object recognition). Specifically, our approach utilizes a neuro-symbolic DSL that leverages pre-trained neural networks for *perception* and higher-level language constructs for *symbolic reasoning*. The combination of these symbolic and neural constructs is very powerful in that it enables (a) reasoning about relationships between different objects in the image, and (b) selectively editing parts of the image that contain some visual cue of interest.

In more detail, a program in our neuro-symbolic DSL is of the form $\{E \to A, \ldots, E \to A\}$, where A is an action and E is an extractor. An action describes a specific image manipulation (e.g. Crop or Blur), and an extractor describes the subset of objects to which that action will be applied. As a simple example, consider the program $\{\mathsf{Object}(\texttt{cat}) \to \mathsf{Brighten}\}$, which applies a brightening filter to all cats in the input image. This DSL also allows combining different objects via standard set operators. For instance, the extractor $\mathsf{Intersect}(\mathsf{Is}(\mathsf{Smiling}), \mathsf{Complement}(\mathsf{Is}(\mathsf{EyesOpen})))$ extracts all human faces that are smiling and do *not* have their eyes open. In addition, a Find operator can be used to extract objects based on their relative position within the image. For instance, the extractor $\mathsf{Find}(\mathsf{Is}(\mathsf{Text}(\texttt{"Total"})), \mathsf{TextObject}, \mathsf{GetRight})$ first identifies all text objects matching the word "Total" and then, for each such object o, it extracts the first text object that is to the right of o.

With these basic DSL constructs in mind, let us consider the program that can be used to automate our target task (i.e., finding and cropping all images that feature the user's daughter with a violin). This task can be expressed with the following program in our DSL:

$$\begin{aligned}\{&\mathsf{Union}(\mathsf{Find}(\mathsf{Is}(\mathsf{Face}(\texttt{Id})), \mathsf{Object}(\texttt{violin}), \mathsf{GetBelow}),\\ &\quad \mathsf{Find}(\mathsf{Is}(\mathsf{Object}(\texttt{violin})), \mathsf{Face}(\texttt{Id}), \mathsf{GetAbove})) \to \mathsf{Crop}\}\end{aligned}$$

Here, the extractor is a Union of two sub-extractors: $\mathsf{Find}(\mathsf{Is}(\mathsf{Face}(\texttt{Id})), \mathsf{Object}(\texttt{Violin}), \mathsf{GetBelow})$ and $\mathsf{Find}(\mathsf{Is}(\mathsf{Object}(\texttt{Violin})), \mathsf{Face}(\texttt{Id}), \mathsf{GetAbove})$. The first sub-extractor identifies all human faces with the identifier Id, where Id corresponds to the face of the user's daughter. Then, for each such face, this program extracts the first violin object located below that face. Conversely, the second sub-extractor identifies all violin objects; then, for each such object, it extracts the first human face with identifier Id that is located above the violin. In other words, the first sub-extractor extracts the violin that is played by the user's daughter, and the second sub-extractor extracts the face of the user's daughter when she is holding a violin. The union of these sub-extractors precisely describes the part of the image that the user wants to select.

Neuro-symbolic program synthesis. To generate the desired program from the user's demonstrations, our approach first represents the training images in *symbolic* form. In particular, rather than

 $\Longrightarrow$ 

Fig. 1. Example input and output.

viewing each image as a set of low-level pixels, our approach generates a symbolic representation of each image, mapping object identifiers to their properties. This symbolic representation is obtained by running pre-trained neural networks for image segmentation and object recognition on the user-provided images.

One of the key advantages of this symbolic image representation is that it allows reducing our complex learning task to the relatively well-understood *programming-by-example (PBE)* problem. In particular, by representing the image in this symbolic form, ImageEye can keep track of which actions have been applied to which objects. Hence, the learning task reduces to synthesizing a so-called *extractor* that can be used to programmatically extract the *desired* objects among *all* the objects comprising the symbolic image.

While our proposed symbolic image representation allows reducing the learning problem to standard PBE, the resulting PBE task is unfortunately quite challenging. In particular, because images often contain a very large number of objects, the search space for the underlying synthesis problem can become quite massive. To make matters worse, each object has a large number of attributes associated with it, where each attribute corresponds to a pre-trained classifier (e.g., for detecting whether someone is smiling, whether an object is a guitar, etc.). Because each attribute corresponds to a built-in function in the underlying DSL, this means that the space of all programs that the synthesizer needs to consider can be enormous.

As described briefly in Section 1, the PBE technique underlying our synthesis engine is based on top-down enumerative search, but it utilizes two novel ideas to deal with the scalability challenges that arise in this setting:

Idea #1: Combining term rewriting with partial evaluation. Despite the large search space, it turns out that many of the programs in our DSL are redundant. For example, consider the partial programs P_1 = Union(Is(Face(Id)), □) and P_2 = Union(□, Is(Face(Id))) where □ indicates a *hole* (i.e., unknown subprogram). Since the Union operator is commutative, any solution to the synthesis problem that is a completion of P_2 will also be a completion of P_1. Thus, we can significantly reduce the search space by detecting such redundant partial programs and pruning them from the search space. To that end, our method uses term rewriting to reduce each partial program to a canonical form and discards all non-canonical expressions when performing the search. This idea can be seen as an instance of *equivalence reduction* explored in prior work [Smith and Albarghouthi 2019].

While the above idea is quite useful in our setting, it is nonetheless *not* sufficient to detect all redundant partial programs of interest. In particular, while two partial programs may not be *always* equivalent, they might still be *observationally equivalent* — that is, they are guaranteed to have the same behavior on the *given* set of input images. To gain more intuition, consider the partial extractor E = Union(Intersect(Is(Smiling), Is(EyesOpen)), □) whose canonical form is itself. However, suppose that the example images provided by the user do not contain any human faces that are both smiling and have their eyes open. Under this assumption, Intersect(Is(Smiling), Is(EyesOpen)) is the empty set, so E simplifies to $\square$. Motivated by this observation, our method combines *partial evaluation* with *term rewriting* to further reduce the search space. In particular, our method partially evaluates incomplete programs on the provided input-output examples before reducing them to a canonical form. Because partial evaluation can greatly simplify incomplete programs, combining term writing with partial evaluation significantly amplifies the pruning power of this technique. We believe this insight (namely, combining partial evaluation with term rewriting) could be similarly powerful in reducing the search space in other synthesis settings beyond our image editing domain.

Idea #2: Goal-directed reasoning via image abstractions. Our synthesis method uses another key idea, namely *goal-directed reasoning via abstraction*, to successfully automate image editing tasks of interest. For example, consider the partial program Intersect($\square_1$, $\square_2$), and suppose the goal output o is the set of all dog objects in the example images. While we cannot infer the *exact* output of each hole, we *can* infer that the $\square_1$ and $\square_2$ *must* both produce all dog objects in the image due to the semantics of set intersection. Using this kind of goal-directed reasoning, we can prune all partial programs where either hole is instantiated with Is(Object(Cat)) (or any other extractor that does not produce all dogs). Similarly, consider the partial program Union($\square_1$, $\square_2$), and suppose that the target output is again the set of all dogs in the image (and nothing else). In this case, we can infer that each hole should *not* produce anything other than a dog because of the semantics of set union. Hence, if either hole is instantiated with Is(Object(Cat)) (or any extractor that produces a non-dog object), we know that the program will be infeasible and can be safely pruned.

Based on this motivation, our synthesis algorithm performs a form of abstract interpretation over images to facilitate goal-directed reasoning. In particular, starting from the desired output image, our synthesis technique utilizes the abstract semantics of the image editing DSL to infer specifications of sub-programs yet to be synthesized. These specifications take the form of pairs of over- and under-approximations, $(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+)$, where $\hat{\mathcal{I}}^+$ includes all objects that *may* be present in the synthesized program and $\hat{\mathcal{I}}^-$ represents those objects that *must* be present. If a sub-program with inferred specification $(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+)$ ever produces a symbolic image that contains fewer objects than its over-approximation or more objects than its under-approximation, we know it must be incorrect. In our setting, this idea can be used to prune large parts of the search space, and we believe that our proposed abstraction could be similarly useful in other program synthesis tasks involving images.

3 DOMAIN-SPECIFIC LANGUAGE FOR IMAGE MANIPULATION

In this section, we introduce our domain-specific language for image manipulations. Since inputs to programs in this DSL are images, we first explain how we represent images and then describe the constructs in this DSL.

Image representation. A raw image $\mathcal{I}$ as a $n \times m$ matrix where each entry corresponds to a pixel. However, because raw images are quite low level, this work utilizes a more abstract representation called a *symbolic image* for formalizing our DSL. Specifically, given a raw image $\mathcal{I}$, we define a corresponding *symbolic image* $\hat{\mathcal{I}}$ as follows:

Definition 3.1. **(Symbolic image)** A *symbolic image* $\hat{\mathcal{I}}$ is a set of *objects o* where each object is represented by a pair (Φ, Δ) where Φ is a mapping from the attributes of that object to their values, and Δ represents the location of the object within the raw image. For simplicity, we represent Δ as a bounding box (j_l, j_r, j_t, j_b) describing the left, right, top and bottom pixels.

$$\hat{\mathcal{I}} = \{(\Phi_1, \Delta_1), (\Phi_2, \Delta_2), (\Phi_3, \Delta_3), (\Phi_4, \Delta_4)\}$$
$\Phi_1 = \{$ `objectType` $\rightarrow$ `person` $\}$
$\Phi_2 = \{$ `objectType` $\rightarrow$ `face`,
`faceId` $\rightarrow$ `1`,
`Smiling` $\rightarrow$ `true`,
`EyesOpen` $\rightarrow$ `true` $\}$
$\Phi_3 = \{$ `objectType` $\rightarrow$ `car` $\}$
$\Phi_4 = \{$ `objectType` $\rightarrow$ `text`,
`textBody` $\rightarrow$ `"FDE945"` $\}$

Fig. 2. An image and its corresponding symbolic image. Rectangles denote bounding boxes.

$$\begin{array}{rl}
P := & \{E \rightarrow A, \cdots, E \rightarrow A\} \\
A := & \mathsf{Blur} \mid \mathsf{Blackout} \mid \mathsf{Sharpen} \mid \mathsf{Brighten} \mid \mathsf{Recolor} \mid \mathsf{Crop} \\
E := & \mathsf{All} \mid \mathsf{Is}(\varphi) \\
\mid & \mathsf{Complement}(E) \mid \mathsf{Union}_N(E_1, \cdots, E_N) \mid \mathsf{Intersect}_N(E_1, \cdots, E_N) \\
\mid & \mathsf{Find}(E, \varphi, f) \mid \mathsf{Filter}(E, \varphi) \\
\varphi := & \mathsf{Face}(N) \mid \mathsf{Object}(O) \mid \mathsf{Smiling} \mid \mathsf{AboveAge}(N) \mid \mathsf{Text}(W) \mid \cdots \\
f := & \mathsf{GetLeft} \mid \mathsf{GetRight} \mid \mathsf{GetAbove} \mid \mathsf{GetBelow} \mid \mathsf{GetParents}
\end{array}$$

Fig. 3. Image manipulation DSL.

Intuitively, an symbolic image $\hat{\mathcal{I}}$ corresponds to a more abstract representation of $\mathcal{I}$ obtained through pre-trained neural networks used for classification (to construct Φ) and segmentation (used to construct Δ). In particular, each element in the domain of Φ is obtained using a different pre-trained neural network used for classification. For example, consider an attribute called `objectType` in the domain of Φ. This attribute identifies the type of the object, which could be a face, cat, dog, table etc. Some of the attributes are only defined for certain types of objects. For example, the boolean attribute `Smiling` only makes sense for human faces. Thus, the domain of Φ may be different across different objects. Given an object $o = (\Phi, \Delta)$, we use the notation $o.\Phi$ and $o.\Delta$ to refer to Φ and Δ respectively.

Example 3.2. Consider the image in Figure 2, and its corresponding symbolic image $\hat{\mathcal{I}}$ on the right. Here, $\hat{\mathcal{I}}$ contains four objects: the person, their face, the car, and the text on the car's license plate. Each object has an attribute called `objectType`. The face object has the additional attributes `faceId`, `Smiling`, and `EyesOpen`, and the text object has the additional attribute `textBody` whose value is a string that contains the text on the license plate.

In the remainder of this paper, we use a single symbolic image to represent multiple raw input images. Because a symbolic image is a mapping from object identifiers to their attributes, a symbolic image can represent multiple raw images without any loss of information. This is because different occurrences of the same object in different images have different identifiers, and an attribute is used to track which object identifier originates from which image. This design choice of representing multiple raw images as a single symbolic image allows simplifying our technical presentation.

DSL Syntax. Our image editing DSL is defined in Figure 3 and is meant to capture a broad class of selective edits . At the top level, a program is comprised of a set of guarded actions of the form $E \rightarrow A$, where A is an action like Crop or Blur and the guard E is an *extractor* that specifies what part of the image to apply that action to. Extractors are defined recursively and have two base cases: (1) the identity extractor All returns the entire image, and (2) $\mathsf{Is}(\varphi)$ returns all objects in the image for which the predicate φ evaluates to true. Extractors can be nested inside one another by composing them via set operators (complement, intersection, and union) as well as the constructs

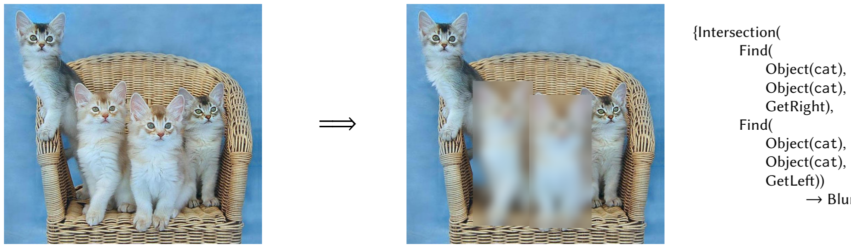

Fig. 4. Input and output of a program.

$$\frac{R \in \{\mathsf{Smiling}, \mathsf{AreEyesOpen}, \cdots\} \quad R \in \mathsf{Domain}(o.\Phi) \quad o.\Phi[R] = \mathsf{True}}{o \models R} \qquad \frac{R \in \{\mathsf{BelowAge}, \mathsf{IsObject}, \cdots\} \quad R \in \mathsf{Domain}(o.\Phi) \quad o.\Phi[R] = C}{o \models R(C)}$$

Fig. 5. Definition of the entailment relation.

$\mathsf{Find}(E, \varphi, f)$ and $\mathsf{Filter}(E, \varphi)$. The Find construct first extracts a set of objects O using the nested extractor E and then, for each object $o \in O$, it returns the first element in $f(o)$ satisfying predicate φ. Here, f is a function that takes as input an object and returns a sorted list of objects. For example, $\mathsf{GetRight}(o)$ returns a list of all objects that are to the right of o, sorted by their x coordinate. Hence, the extractor $\mathsf{Find}(\mathsf{Is}(\mathsf{Face}(n)), \mathsf{Smiling}, \mathsf{GetRight})$ finds the first smiling face to the right of person n. As another example, $\mathsf{Find}(\mathsf{All}, \mathsf{Smiling}, \mathsf{GetLeft})$ would yield the set of all smiling faces that are to the left of *some* object in the input image. Finally, the construct $\mathsf{Filter}(E, \varphi)$ filters nested objects that satisfy predicate φ. In particular, given a set of objects O extracted via E, $\mathsf{Filter}(E, \varphi)$ returns all objects satisfying φ contained inside some object o in O. For instance, the extractor $\mathsf{Filter}(\mathsf{Is}(\mathsf{Object}(\mathtt{car})), \mathsf{Object}(\mathtt{person}))$ will return all people who are inside of cars.

Example 3.3. Consider the image on the left in Figure 4. Given this image as input, the program

$$\begin{array}{l}\{\mathsf{Intersection}(\\ \qquad \mathsf{Find}(\mathsf{Object}(\mathtt{cat}), \mathsf{Object}(\mathtt{cat}), \mathsf{GetRight}),\\ \qquad \mathsf{Find}(\mathsf{Object}(\mathtt{cat}), \mathsf{Object}(\mathtt{cat}), \mathsf{GetLeft})) \rightarrow \mathsf{Blur}\}\end{array}$$

will output the image on the right. Note that the extractor in this program yields all cat objects that have a cat to their left and right. In other words, it extracts all cats that are between two other cats.

Predicates in our DSL reflect the capabilities of state-of-the-art computer vision models for object recognition and classification. In particular, we choose to include certain predicates, such as $\mathsf{Face}(n)$ or $\mathsf{Object}(\mathtt{cat})$, but not others (e.g., Angry, Sad), because existing neural networks are good at detecting the first class of features but not the latter class. Furthermore, the choice of built-in functions (e.g., GetLeft, GetAbove) is motivated by performing segmentation at the level bounding boxes. The remaining constructs are either standard set operations (e.g., Union) or well-understood functional combinators (e.g., Filter).

DSL Semantics. The formal semantics of this DSL are presented in Figure 6. Given a program P and input image $\mathcal{I}$, $[\![P]\!](\mathcal{I})$ produces a new image $\mathcal{I}'$ by applying each of the actions in P to the extracted sub-image. Similarly, given an extractor E and symbolic image $\hat{\mathcal{I}}$, $[\![E]\!](\hat{\mathcal{I}})$ returns a set of objects contained in $\hat{\mathcal{I}}$. Because each object stores its corresponding pixels in the original input image, note that it is trivial to convert a set of objects to pixels of the original image.

The semantics of extractors are defined in terms of symbolic images introduced in Definition 3.1. In particular, given an image $\mathcal{I}$, $\mathsf{Is}(\varphi)$ returns the set of all objects in $\hat{\mathcal{I}}$ that satisfy φ. As defined in Figure 5, an object o satisfies a predicate of the form $R(C)$, denoted $o \models R(C)$, if $o.\Phi$ contains an

$$
\begin{array}{rl}
[\![E \rightarrow A]\!](\mathcal{I}) = & \text{ApplyAction}(\mathcal{I}, A, \bigcup_{o \in [\![E]\!](\hat{\mathcal{I}})} o.\Delta) \\
[\![E \rightarrow A,\ P]\!](\mathcal{I}) = & \text{ApplyAction}([\![P]\!](\mathcal{I}), A, \bigcup_{o \in [\![E]\!](\hat{\mathcal{I}})} o.\Delta) \\
[\![\text{All}]\!](\hat{\mathcal{I}}) = & \hat{\mathcal{I}} \\
[\![\text{Is}(\varphi)]\!](\hat{\mathcal{I}}) = & \{o \mid o \in \hat{\mathcal{I}} \land o \models \varphi\} \\
[\![\text{Union}(\overline{E})]\!](\hat{\mathcal{I}}) = & \bigcup_{E \in \overline{E}} [\![E]\!](\hat{\mathcal{I}}) \\
[\![\text{Intersect}(\overline{E})]\!](\hat{\mathcal{I}}) = & \bigcap_{E \in \overline{E}} [\![E]\!](\hat{\mathcal{I}}) \\
[\![\text{Complement}(E)]\!](\hat{\mathcal{I}}) = & \hat{\mathcal{I}} \setminus [\![E]\!](\hat{\mathcal{I}}) \\
[\![\text{Filter}(E, \varphi)]\!](\hat{\mathcal{I}}) = & \texttt{flatten}(\texttt{map}([\![E]\!](\hat{\mathcal{I}}), \lambda x.\ \texttt{filter}([\![\text{GetContents}]\!](x, \hat{\mathcal{I}}), \varphi)))) \\
[\![\text{Find}(E, \varphi, f)]\!](\hat{\mathcal{I}}) = & \texttt{map}([\![E]\!](\hat{\mathcal{I}}), \lambda x.\ [\![f_\varphi(x)]\!](\hat{\mathcal{I}})) \\
[\![f_\varphi(o)]\!](\hat{\mathcal{I}}) = & \begin{cases} S[i] & \texttt{if } \exists_{0 \le i < |S|} S[i] \models \varphi \bigwedge \forall_{0 \le j < i}\, S[j] \not\models \varphi \\ \text{None} & \texttt{otherwise} \end{cases} \quad \texttt{where } S = [\![f]\!](o, \hat{\mathcal{I}})
\end{array}
$$

Fig. 6. DSL semantics. Here, $\texttt{map}(S, f)$ takes the input $S : \text{Set}[T], f : T \rightarrow \text{Option}[T]$ and returns $\{f(s) \mid s \in S \land f(s) \neq \text{None}\}$. `flatten` takes in a set of sets $S_{\text{All}} = \{S_1, \cdots S_n\}$ and returns a set that containing $\bigcup_{S_i \in S_{\text{All}}} \cup_{s \in S_i} s$. Finally, `filter` is the standard filter operator.

$$
\begin{array}{rl}
[\![\text{GetRight}]\!](o, \hat{\mathcal{I}}) = & \text{Sort}(\{o' \mid o' \in \hat{\mathcal{I}} \land o'.\Delta[j_\text{l}] \ge o.\Delta[j_\text{l}]\}, o.\Delta[j_\text{l}]) \\
[\![\text{GetLeft}]\!](o, \hat{\mathcal{I}}) = & \text{SortReverse}(\{o' \mid o' \in \hat{\mathcal{I}} \land o'.\Delta[j_\text{r}] \le o.\Delta[j_\text{r}]\}, o.\Delta[j_\text{r}]) \\
[\![\text{GetAbove}]\!](o, \hat{\mathcal{I}}) = & \text{Sort}(\{o' \mid o' \in \hat{\mathcal{I}} \land o'.\Delta[j_\text{t}] \ge o.\Delta[j_\text{t}]\}, o.\Delta[j_\text{t}]) \\
[\![\text{GetBelow}]\!](o, \hat{\mathcal{I}}) = & \text{SortReverse}(\{o' \mid o' \in \hat{\mathcal{I}} \land o'.\Delta[j_\text{b}] \ge o.\Delta[j_\text{b}]\}, o.\Delta[j_\text{b}]) \\
[\![\text{GetParents}]\!](o, \hat{\mathcal{I}}) = & \text{Sort}(\{o' \mid o' \in \hat{\mathcal{I}} \land [\![\text{Contains}]\!](o'.\Delta, o.\Delta)\}, \text{GetSize}(\Delta)) \\
[\![\text{GetContents}]\!](o, \hat{\mathcal{I}}) = & [o' \mid o' \in \hat{\mathcal{I}} \land [\![\text{Contains}]\!](o.\Delta, o'.\Delta) \land \\
& \nexists_{o'' \in \hat{\mathcal{I}}}.\ o'' \neq o \land [\![\text{Contains}]\!](o.\Delta, o''.\Delta) \land [\![\text{Contains}]\!](o''.\Delta, o'.\Delta)] \\
[\![\text{Contains}]\!](o.\Delta, o'.\Delta) = & \text{True if} \\
& o'.\Delta[j_\text{l}] \ge o.\Delta[j_\text{l}] \land o'.\Delta[j_\text{r}] \le o.\Delta[j_\text{r}] \\
& \land o'.\Delta[j_\text{t}] \ge o.\Delta[j_\text{t}] \land o'.\Delta[j_\text{b}] \le o.\Delta[j_\text{b}] \text{ else False} \\
[\![\text{GetSize}]\!](\Delta) = & (\Delta[j_\text{r}] - \Delta[j_\text{l}]) * (\Delta[j_\text{b}] - \Delta[j_\text{t}])
\end{array}
$$

Fig. 7. Semantics for the built-in and auxiliary functions f in the DSL. Sort(S, key) sorts the objects in set S from smallest to largest with respect to key. SortReverse does the opposite.

attribute called R and the value of that attribute is C. Similarly, if R is a nullary relation, we have $o \models R$ iff o has an attribute called R whose value is *true*.

Since the semantics of set operators are standard, we only explain the semantics of Filter and Find, which are defined in terms of functional combinators like `map` and `flatten`. Recall that the Find extractor is parameterized over a function f, such as GetRight and GetBelow, whose semantics are given in Figure 7. In particular, $[\![\text{GetX}]\!](o, \hat{\mathcal{I}})$ yields a list of all objects o' in $\hat{\mathcal{I}}$ satisfying the *spatial* relationship $X(o', o)$. As expected, the semantics of these functions are defined using the bounding box Δ of each object in the image. For example, given an object o in $\hat{\mathcal{I}}$, GetRight decides which objects are to the right of o based on the leftmost pixels of the bounding box of each object.

As shown in Figure 6, the Find construct first evaluates its nested extractor E on the input image $\hat{\mathcal{I}}$ to obtain a set of objects O and applies the function f_φ to each object $o \in O$. The semantics of $f_\varphi(x)$ are given at the very bottom of Figure 6 and essentially yield the first object satisfying φ in the list given by $f(x)$. Since $f(x)$ may be the empty list or may not have any elements satisfying φ, observe that $f_\varphi(x)$ can yield None, which is discarded when constructing the output of Find.

Finally, we explain the semantics of the Filter construct, which first evaluates its nested extractor E on the input image $\hat{\mathcal{I}}$ to obtain a set of elements of O. Then, for each object $o \in O$, it obtains elements nested inside of o (by inspecting the bounding box of each object in the image) and only

procedure SYNTHESIZE(Ψ)
  input: specification Ψ
  output: a program P such that $\forall(\mathcal{I}, \xi) \in \Psi.\ P(\mathcal{I}) = \xi[\mathcal{I}]$
  $P \leftarrow \emptyset$
  for all $A \in$ Actions **do**
    $\hat{\mathcal{I}}_{in} \leftarrow \bigcup_{(\mathcal{I},\xi)\in\Psi} \hat{\mathcal{I}}$
    $\hat{\mathcal{I}}_{out} \leftarrow \{o \mid (\mathcal{I}, \xi) \in \Psi \wedge o \in \text{Domain}(\xi) \wedge A \in \xi[o]\}$
    if $\hat{\mathcal{I}}_{out} \neq \emptyset$ **then**
      $E \leftarrow$ SYNTHESIZEEXTRACTOR($\hat{\mathcal{I}}_{in}, \hat{\mathcal{I}}_{out}$)
      if $E = \perp$ **then**
        return $\perp$
      else
        $P \leftarrow P \cup \{E \rightarrow A\}$
  return P

Fig. 8. Top-level synthesis algorithm

retains those elements that satisfy φ. The final output of Filter is obtained by flattening the resulting set of sets into a single set.

4 PROBLEM STATEMENT

In this section, we define the synthesis problem that we address in the remainder of the paper. We first start by introducing the concept of an *image edit*:

Definition 4.1. **(Edit)** Given an image $\mathcal{I}$, an edit ξ on that image is a mapping from objects in $\hat{\mathcal{I}}$ to a list of actions that have been applied to those objects.

Given an image $\mathcal{I}$ and edit ξ, we use the notation $\mathcal{I}[\xi]$ to denote the resulting image obtained by applying ξ to $\mathcal{I}$. The specification for our synthesis problem is defined in terms of edits:

Definition 4.2. **(Spec)** An *image manipulation specification* Ψ is a mapping from images to edits.

We now formally state our synthesis problem as follows:

Definition 4.3. **(Image manipulation by demonstration (IMBD))** Given an image manipulation specification Ψ, the goal of *image manipulation by demonstration* is to produce a program P in the DSL from Figure 3 such that $\forall(\mathcal{I}, \xi) \in \Psi.\ P(\mathcal{I}) = \xi[\mathcal{I}]$.

5 SYNTHESIS ALGORITHM

In this section, we describe our synthesis algorithm for solving the IMBD problem defined in the previous section. Our top-level learning procedure is shown in Figure 8 and works as follows. For each possible action in the DSL, it constructs an input-output example $(\hat{\mathcal{I}}_{in}, \hat{\mathcal{I}}_{out})$ based on the specification Ψ. If, for some action A, $\hat{\mathcal{I}}_{out}$ is the empty set, this means that A is irrelevant to the target task, so the algorithm moves on to the next action. Otherwise, it invokes the SYNTHESIZEEXTRACTOR procedure on $(\hat{\mathcal{I}}_{in}, \hat{\mathcal{I}}_{out})$ to learn the corresponding extractor E for A and adds the guarded action $E \rightarrow A$ to the synthesized program.

5.1 Preliminaries

As is evident from this discussion, the central part of our technique is the extractor learning algorithm, which relies on a particular representation of *partial programs*:

Definition 5.1. **(Partial program)** A partial program P is a tree (V, E, Σ, Π) with nodes V (including a special root node v_0) and directed edges E. The mapping Σ maps each node in V to a label, which is either a construct in our image manipulation DSL (e.g., All, Complement) or the special symbol $\square$ representing a hole. Each node $v \in V$ is also annotated with a *goal* ϕ such that $\Pi(v) = \phi$. We write $P \vdash v : (l, \phi)$ to denote that $\Sigma(v) = l$ and $\Pi(v) = \phi$. If none of the nodes in P is labeled with a hole, we refer to P as a *complete program.*

Example 5.2. Figure 10 depicts the partial program Union(Is(Smiling), $\square$) as a tree, where each node is annotated with its corresponding label.

The goal annotation $\Pi(v)$ for each node v in a partial program imposes constraints on the semantics of the subtree rooted at v. In our context, a goal is defined as follows:

Definition 5.3. **(Goal annotation)** A *goal annotation* (or *goal* for short) of a node in the partial program is a pair $(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+)$ where $\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+$ are symbolic images corresponding to over- and under-approximations of the output.

Next, we define the consistency between a symbolic image and a goal as follows:

Definition 5.4. **(Consistency with goal)** We say that a symbolic image $\hat{\mathcal{I}}$ is consistent with a goal $\phi = (\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+)$, denoted $\hat{\mathcal{I}} \sim \phi$, iff $\hat{\mathcal{I}}^- \subseteq \hat{\mathcal{I}} \subseteq \hat{\mathcal{I}}^+$.

We also extend this notion of consistency to partial programs:

Definition 5.5. **(Consistency of partial program)** A partial program P is consistent with a symbolic image $\hat{\mathcal{I}}$ iff, for every complete subtree P_v of P rooted at node v, we have $[\![P_v]\!](\hat{\mathcal{I}}) \sim \Pi(v)$.

Intuitively, the goals annotating a partial program are used for guiding extractor synthesis and for ensuring that we never enumerate inconsistent partial programs.

Example 5.6. Consider again the partial program from Figure 10, which contains a complete subprogram, namely Is(Smiling), rooted at node v_1. Suppose that we have an input-output example $(\hat{\mathcal{I}}_{in}, \hat{\mathcal{I}}_{out})$, where $\hat{\mathcal{I}}_{in}$ contains several face objects. Then $[\![\text{Is}(\text{Smiling})]\!](\hat{\mathcal{I}}_{in})$ will be a symbolic image $\hat{\mathcal{I}}'$ containing just the faces that are smiling. Further, suppose our output symbolic image $\hat{\mathcal{I}}_{out}$ contains just the faces that are smiling or have their eyes open. The goal annotation of Is(Smiling) will be $\Pi(v_1) = (\emptyset, \hat{\mathcal{I}}_{out})$, as explained later in Section 5.3. Since $\emptyset \subseteq \hat{\mathcal{I}}' \subseteq \hat{\mathcal{I}}_{out}$, we have $[\![\text{Is}(\text{Smiling})]\!](\hat{\mathcal{I}}_{in}) \sim \Pi(v_1)$. Therefore, this partial program is consistent. To illustrate inconsistent partial programs, now consider Union(Is(Object(cat)), $\square$) and the same input-output example $(\hat{\mathcal{I}}_{in}, \hat{\mathcal{I}}_{out})$. Here, $[\![\text{Is}(\text{Object}(\texttt{cat}))]\!](\hat{\mathcal{I}}_{in})$ will be a symbolic image $\hat{\mathcal{I}}'$ containing all cat objects. The goal annotation of Is(Object(cat)) will again be $\Pi(v_1) = (\emptyset, \hat{\mathcal{I}}_{out})$. Since $\hat{\mathcal{I}}' \nsubseteq \hat{\mathcal{I}}_{out}$, $[\![\text{Is}(\text{Object}(\texttt{cat}))]\!](\hat{\mathcal{I}}_{in}) \nsim \Pi(v_1)$. Therefore, this partial program is inconsistent.

Because our synthesis algorithm gradually replaces holes with concrete programs, we conclude this section by defining an operation to update partial programs:

Definition 5.7. **(Partial program update)** Given a partial program $P = (V, E, \Sigma, \Pi)$, we use the notation $P[v_0 \triangleleft (l_0, \phi_0), \ldots v_n \triangleleft (l_n, \phi_n)]$ to indicate the new partial program

$$P' = (V \cup \bigcup_{i=0}^{n} \{v_i\},\ E \cup \bigcup_{i=1}^{n} \{(v_0, v_i)\}, \Sigma[v_0 \mapsto l_0, \ldots v_n \mapsto l_n], \Pi[v_0 \mapsto l_0, \ldots v_n \mapsto l_n])$$

In other words, the notation $P[v_0 \triangleleft (l_1, \phi_1), \ldots v_n \triangleleft (l_n, \phi_n)]$ corresponds to adding children $(v_1, \ldots, v_n)$ of v_0 (and their corresponding labels and goals) and updating the label and goal of v_0. Finally, we write CreateProg(v, l, ϕ) to denote the creation of a partial program with a single node v with label l and goal annotation ϕ.

1: **procedure** SynthesizeExtractor($\hat{\mathcal{I}}_{in}, \hat{\mathcal{I}}_{out}$)
input: $\hat{\mathcal{I}}_{in}$ is an input symbolic image and $\hat{\mathcal{I}}_{out}$ is the output symbolic image
output: an extractor program P such that $[\![P]\!](\hat{\mathcal{I}}_{in}) \equiv \hat{\mathcal{I}}_{out}$
2: $\mathcal{W} \leftarrow \{\text{CreateProg}(v_0, \square, (\hat{\mathcal{I}}_{out}, \hat{\mathcal{I}}_{out}))\}$
3: **while** $\mathcal{W} \neq \emptyset$ **do**
4: $P \leftarrow \mathcal{W}.\text{remove}()$
5: **if** isComplete(P) **then**
6: **if** $[\![P]\!](\hat{\mathcal{I}}_{in}) \equiv \hat{\mathcal{I}}_{out}$ **then return** P
7: **else**
8: $v \leftarrow$ SelectOpenNode(P);
9: **for all** $P' \in$ Expand(P, v) **do**
10: $P'' \leftarrow$PartialEval($P', \hat{\mathcal{I}}_{in}$)
11: **if** $P'' \neq \bot \wedge \neg$Reducible($P''$) **then**
12: $\mathcal{W} \leftarrow \mathcal{W} \cup \{P'\}$;
13: **return** $\bot$

Fig. 9. Extractor synthesis algorithm.

5.2 Top-Level Extractor Learning Algorithm

We now present our top-level extractor learning algorithm, which is shown in Figure 9. Given an input symbolic image $\hat{\mathcal{I}}_{in}$ and an output symbolic image $\hat{\mathcal{I}}_{out}$, this algorithm maintains a worklist $\mathcal{W}$ of partial programs and iteratively adds to this list. At the beginning of the procedure, $\mathcal{W}$ is initialized to a single program with one node v_0 and no edges. Node v_0 has label $\square$ and goal output $\phi_0 = (\hat{\mathcal{I}}_{out}, \hat{\mathcal{I}}_{out})$. The loop in lines 3-12 dequeues a program P from the worklist and processes it. The worklist keeps programs in ascending order first by AST size, then by AST depth. If P is complete and satisfies the correctness condition, the procedure terminates and returns P. Otherwise, SynthesizeExtractor calls Expand on line 9 to generate a new set of partial programs by expanding an open node v in P. As we describe in Section 5.3, the expansion procedure also infers goals for each new hole in the partial program.

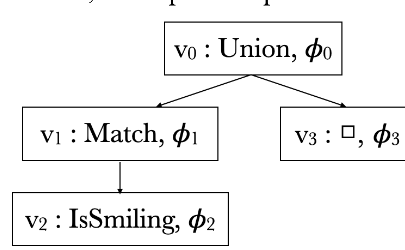

Fig. 10. An illustration of a partial program.

Next, for each expansion P' of P, the algorithm calls PartialEval on line 10 to generate a partially evaluated program P''. The partial evaluation procedure identifies each complete subprogram P_v of P', evaluates it on the input image $\hat{\mathcal{I}}_{in}$ to obtain an output image $\hat{\mathcal{I}}$, and replaces P_v with the constant $\hat{\mathcal{I}}$. As we discuss in more detail in Section 5.4, PartialEval can return $\bot$ if it finds any inconsistent subprograms; in this case, P' is *not* added to the worklist.

If partial evaluation does not return $\bot$, the algorithm calls the Reducible procedure on line 11, which is used to check whether P'' can be simplified. Since SynthesizeExtractor explores programs in increasing order of complexity, we know that P'' is redundant if Reducible returns $\top$. Hence, the algorithm adds P'' to the worklist only if P'' cannot be further simplified.

5.3 Goal Inference

As mentioned earlier, a key component of our extractor learning approach is the inference of goals for each node. This goal inference method is presented in Figure 11 as part of the Expand procedure. Every time the algorithm expands a hole associated with node v, it picks an (n-ary) DSL operator f, updates v's label to f, and adds n new children $v_1, \ldots, v_n$ of v. Each child node v_i of v is marked as being "open" (i.e., labeled with a hole) and is annotated with its corresponding goal. Observe that

$$\frac{P \vdash v : (\square, \phi) \qquad \phi_f = \llfloor \mathsf{f} \rrceil(\phi)}{\textsc{Expand}(P, v) = \{P[v \triangleleft (\mathsf{f}, \phi), v_1 \triangleleft (\square, \phi_f), \dots, v_n \triangleleft (\square, \phi_f)] \mid \mathsf{f} \in \mathcal{F}, \overline{v} \text{ fresh}\}}$$

$$\begin{aligned}
\llfloor \mathsf{Union} \rrceil(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+) &= (\emptyset, \hat{\mathcal{I}}^+) \\
\llfloor \mathsf{Intersect} \rrceil(\hat{\mathcal{I}}^- \hat{\mathcal{I}}^+) &= (\hat{\mathcal{I}}^-, \hat{\mathcal{I}}_{in}) \\
\llfloor \mathsf{Complement} \rrceil(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+) &= (\hat{\mathcal{I}}_{in} \setminus \hat{\mathcal{I}}^+, \hat{\mathcal{I}}_{in} \setminus \hat{\mathcal{I}}^-) \\
\llfloor \mathsf{FilterContents} \rrceil(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+) &= (\emptyset, \hat{\mathcal{I}}_{in}) \\
\llfloor \mathsf{Find} \rrceil(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+) &= (\emptyset, \hat{\mathcal{I}}_{in})
\end{aligned}$$

Fig. 11. Inference rules for Expand. $\hat{\mathcal{I}}_{in}$ is the input symbolic image, and $\mathcal{F}$ represents DSL functions.

goal inference is performed using the function $\llfloor \mathsf{f} \rrceil$, which takes as input a goal annotation ϕ and produces a new goal ϕ_f for the arguments of the function f.

Recall that a goal annotation is of the form $(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+)$ where $\hat{\mathcal{I}}^-$ and $\hat{\mathcal{I}}^+$ are symbolic images under- and over-approximating the image objects associated with the subprogram rooted at that node. In more detail, if a node v has the goal annotation $(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+)$, the consistency requirement from Definition 5.5 stipulates that, in order for P to be consistent with the input symbolic image $\hat{\mathcal{I}}_{in}$, the subprogram P_v rooted at node v must produce a set of objects that is a superset of $\hat{\mathcal{I}}^-$ when executed on the input image $\hat{\mathcal{I}}_{in}$. Similarly, it also requires that $[\![P_v]\!](\hat{\mathcal{I}}_{in})$ is a subset of $\hat{\mathcal{I}}^+$. Hence, given a goal ϕ on the output of a DSL operator f, goal inference aims to propagate under- and over-approximations to each of f's arguments. Put simply, the goal annotations approximate the output that a program must have in order for its parent program to also have a valid output, so programs that do not match their goal annotation can be safely pruned from the search space. We formalize this notion in the following theorem.[1]

Theorem 5.8. *Let P be a partial program derived by SynthesizeExtractor whose root node has goal annotation $(\hat{\mathcal{I}}_{out}, \hat{\mathcal{I}}_{out})$. If P is not consistent with $\hat{\mathcal{I}}_{in}$, then for any completion P' of P, $[\![P']\!](\hat{\mathcal{I}}_{in}) \not\equiv \hat{\mathcal{I}}_{out}$.*

The proof of this theorem crucially relies on the correctness of the goal inference rules, which we explain in more detail next.

Union. Consider the DSL expression $E = \mathsf{Union}(E_1, \dots, E_n)$, and suppose that the goal annotation for this expression is $(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+)$. Since the over-approximation for the whole expression is $\hat{\mathcal{I}}^+$, the operands of Union should not produce objects that are not in $\hat{\mathcal{I}}^+$. Hence, the over-approximation for each operand is also $\hat{\mathcal{I}}^+$. In other words, if *any* operand outputs an object o that is not in $\hat{\mathcal{I}}^+$, then E will output o as well, which is not valid. In contrast, the only safe under-approximation we can infer for the operands is $\emptyset$, as there is no particular object o that each operand *must* output in order for E to output all objects in $\hat{\mathcal{I}}^-$.

Intersect. Consider the expression $E = \mathsf{Intersect}(E_1, \dots, E_n)$, and suppose that the goal annotation for this expression is $(\hat{\mathcal{I}}^-, \hat{\mathcal{I}}^+)$. By the semantics of Intersect, for E to output each $o \in \hat{\mathcal{I}}^-$, each operand E_i must also output o. Thus, the under-approximation for each E_i is also $\hat{\mathcal{I}}^-$. In contrast, we cannot deduce anything about elements that must *not* be in the output of any E_i; thus, the over-approximation for the operands is the entire input image $\hat{\mathcal{I}}_{in}$.

Complement. Consider the expression $E = \mathsf{Complement}(E')$. If E must produce an object o (i.e., $o \in \hat{\mathcal{I}}^-$), then E' must not produce it. Hence, the over-approximation for E' is $\hat{\mathcal{I}}_{in} \setminus \hat{\mathcal{I}}^-$. In contrast,

[1] Proofs are provided in the Appendix of the extended version of the paper [Barnaby et al. 2023b].

$$\textsc{Hole}\ \frac{\mathsf{Root}(P) = v \qquad P \vdash v : (\square, \phi)}{\hat{\mathcal{I}}_{in} \vdash P \rightsquigarrow P}$$

$$\textsc{Const}\ \frac{\mathsf{Root}(P) = v \qquad P \vdash v : (\hat{\mathcal{I}}, \phi)}{\hat{\mathcal{I}}_{in} \vdash P \rightsquigarrow P}$$

$$\textsc{Complete}\ \frac{\mathsf{IsComplete}(P) \qquad [\![P]\!](\hat{\mathcal{I}}_{in}) = \hat{\mathcal{I}} \qquad \mathsf{Root}(P) = v \qquad P \vdash v : (l, \phi)}{\hat{\mathcal{I}}_{in} \vdash P \rightsquigarrow (\hat{\mathcal{I}} \sim \phi)\ ?\ \mathsf{CreateProg}(v, \hat{\mathcal{I}}, \phi) : \bot}$$

$$\textsc{Partial}\ \frac{\begin{array}{c}\neg\mathsf{IsComplete}(P) \qquad \mathsf{Root}(P) = v \qquad P \vdash v : (l, \phi) \\ \mathsf{Children}(P, v) = \{v_1, \dots, v_n\} \qquad \hat{\mathcal{I}}_{in} \vdash \mathsf{Subtree}(P, v_i) \rightsquigarrow P_i\end{array}}{\hat{\mathcal{I}}_{in} \vdash P \rightsquigarrow (\forall_i.\ P_i \neq \bot)\ ?\ P[P_1/\mathsf{Subtree}(P, v_1), \cdots, P_n/\mathsf{Subtree}(P, v_n)] : \bot}$$

Fig. 12. Rules for PartialEvaluation. $P[P_i/\mathsf{Subtree}(P, v_i)]$ represents replacing the subprogram of P rooted at node v_i with the new subprogram P_i.

if E must not produce an object o (i.e., $o \notin \hat{\mathcal{I}}^+$), then o must be produced by E'. Hence, the under-approximation is $\hat{\mathcal{I}}_{in} \backslash \hat{\mathcal{I}}^+$.

Find, Filter. In the case of the Find and Filter constructs, we cannot propagate meaningful approximations to the nested extractors, resulting in the trivial goal annotation $(\emptyset, \hat{\mathcal{I}}_{in})$. To gain intuition about why this is the case, consider the expression $\mathsf{Find}(E', \varphi, \mathsf{GetLeft})$. For any object $o \in \hat{\mathcal{I}}_{in}$, it could be the case that there is no object located to the left of o in image $\mathcal{I}_{in}$, meaning that $[\![\mathsf{GetLeft}]\!](o, \hat{\mathcal{I}}_{in})$ will be empty. In other words, if o is output by E', it will have no impact on the output of E. Hence, any object in $\hat{\mathcal{I}}_{in}$ *could* be output by E', which is why the over-approximation is $\hat{\mathcal{I}}_{in}$. For similar reasons, we also cannot infer any sound under-approximation other than $\emptyset$.

Example 5.9. Consider the image $\hat{\mathcal{I}}$ from Figure 2, and let $\hat{\mathcal{I}}_{out} = \{(\Phi_4, \Delta_4)\}$ be the output symbolic image containing only the license plate. Now consider the partial program:

$$\mathsf{Union}(\mathsf{Complement}(\mathsf{Is}(\mathsf{Object}(\mathtt{car}))), \square)$$

whose top-level goal is $(\hat{\mathcal{I}}_{out}, \hat{\mathcal{I}}_{out})$. Since the subprogram $\mathsf{Complement}(\mathsf{Is}(\mathsf{Object}(\mathtt{car})))$ is an operand of a Union, it has goal $(\emptyset, \hat{\mathcal{I}}_{out})$. Further, the subprogram $\mathsf{Is}(\mathsf{Object}(\mathtt{car}))$ is the operand of a Complement, so it has goal $(\{(\Phi_1, \Delta_1), (\Phi_2, \Delta_2), (\Phi_3, \Delta_3)\}, \hat{\mathcal{I}})$.

5.4 Partial Evaluation

As stated earlier, our synthesis algorithm performs partial evaluation to amplify the power of goal-directed reasoning as well as equivalence reduction. In particular, given a partial program P, the PartialEval procedure invoked in Figure 9 returns another partial program P' by evaluating the complete subprograms of P on the input. Partial evaluation can also reveal that P is infeasible; in this case, PartialEval returns $\bot$ to indicate that P violates consistency (Definition 5.5).

We present our PartialEval procedure using the inference rules summarized in Figure 12. The first rule, labeled Hole, states that open nodes cannot be evaluated, as they represent a completely unconstrained program. The second rule, labeled Const, states that constants simply evaluate to themselves. The third rule, labeled Complete, evaluates complete subprograms by executing them on the input. If the resulting output $\hat{\mathcal{I}}$ is inconsistent with the goal annotation ϕ, partial evaluation yields $\bot$; otherwise, it produces the constant $\hat{\mathcal{I}}$. The final rule, labeled Partial, applies to incomplete programs and recursively applies PartialEval to each subprogram rooted at the root

$$
\begin{array}{rl}
\mathsf{Union}(A, A) \rightsquigarrow A & \mathsf{Intersect}(A, A) \rightsquigarrow A \\
\mathsf{Union}(A, \mathsf{Intersect}(A, B)) \rightsquigarrow A & \mathsf{Intersect}(A, \mathsf{Union}(A, B)) \rightsquigarrow A \\
\mathsf{Union}(A, B) \rightsquigarrow B \text{ if } A \subseteq B. & \mathsf{Intersect}(A, B) \rightsquigarrow A \text{ if } A \subseteq B. \\
\mathsf{Complement}(\mathsf{Complement}(A)) \rightsquigarrow A & \mathsf{Union}(B, A) \rightsquigarrow \mathsf{Union}(A, B) \\
\mathsf{Intersect}(B, A) & \rightsquigarrow \mathsf{Intersect}(A, B) \\
\mathsf{Union}(\mathsf{Complement}(A), \mathsf{Complement}(B)) & \rightsquigarrow \mathsf{Complement}(\mathsf{Intersect}(A, B)) \\
\mathsf{Intersect}(\mathsf{Complement}(A), \mathsf{Complement}(B)) & \rightsquigarrow \mathsf{Complement}(\mathsf{Union}(A, B)) \\
\mathsf{Union}(\mathsf{Intersect}(A, B), \mathsf{Intersect}(A, C)) & \rightsquigarrow \mathsf{Intersect}(A, \mathsf{Union}(B, C)) \\
\mathsf{Intersect}(\mathsf{Union}(A, B), \mathsf{Union}(A, C)) & \rightsquigarrow \mathsf{Union}(A, \mathsf{Intersect}(B, C))
\end{array}
$$

Fig. 13. Rewrite rules.

node. If any of these subprograms are inconsistent, then the whole program is also inconsistent, and the algorithm returns $\perp$. Otherwise, it constructs a new partial program where each subprogram P_i of the root node is replaced with its partially evaluated version P_i'.

Example 5.10. Consider the program $\mathsf{Union}(\mathsf{Complement}(\mathsf{Is}(\mathsf{Object}(\mathsf{car}))), \square)$ from Example 5.9 and the desired output image containing just the license plate (i.e., $\{(\Phi_4, \Delta_4)\}$). This program is incomplete, so the Partial rule will recursively apply PartialEval. The subprogram $\mathsf{Complement}(\mathsf{Is}(\mathsf{Object}(\mathsf{car}))$ is complete, so the Complete rule will evaluate this subprogram on the input symbolic image $\hat{\mathcal{I}}$ to obtain $\hat{\mathcal{I}}' = \{(\Phi_1, \Delta_1), (\Phi_2, \Delta_2), (\Phi_4, \Delta_4)\}$. Recall also (from Example 5.9) that the goal of this subprogram is $(\emptyset, \{(\Phi_4, \Delta_4)\})$. Since $\hat{\mathcal{I}}' \not\subseteq \{(\Phi_4, \Delta_4)\}$, $\hat{\mathcal{I}}'$ is not consistent with the goal, so partial evaluation will return $\perp$. Intuitively, this program should be pruned because, no matter how we instantiate the hole, the top-level program will always produce objects (e.g., the human face) that are not part of the desired output image.

5.5 Equivalence Reduction

We conclude this section by describing our equivalence reduction technique for identifying redundant partial programs. In particular, recall that a partial program P' is *redundant* with respect to another partial program P if, for every completion C' of P', there is a corresponding completion C of P such that C and C' produce the same output on the input examples. In other words, because such partial programs P, P' are observationally equivalent on the inputs of interest, it suffices to merge them into one equivalence class. Thus, our technique can be viewed as extending the notion of *observational equivalence* from complete to partial programs.

At a high level, there are two key components of our equivalence reduction technique: (1) partial evaluation (already discussed in Section 5.4) and (2) term rewriting. Given a partially evaluated program P, our synthesis algorithm checks whether it is possible to simplify P using a set of rewrite rules that capture known equivalences between expressions in our DSL. Figure 13 shows the rewrite rules for our DSL using the notation $l \rightsquigarrow r$, meaning that a term that matches l can be rewritten into the form on the right. Observe that the free variables in l and r are universally quantified, so a term t is said to match the left-hand-side l if there exists a substitution σ such that $t = l[\sigma]$. Furthermore, the result of applying this rewrite rule to t is $r[\sigma]$.

With this notation in place, we now turn our attention to the Reducible procedure called by the SynthesizeExtractor algorithm. Recall that Reducible returns a boolean ($\top$ or $\perp$) to indicate whether a term can be simplified using a set Ω of domain-specific rewrite rules. This Reducible procedure is defined using the two inference rules shown in Figure 14. According to the first rule (Base), holes and constant values are not reducible. The second rule labeled Rec deals with terms $E \equiv \mathsf{f}(E_1, \ldots, E_n)$ by recursively invoking the Reducible procedure on each E_i. If any E_i is reducible, it also returns reducible. Otherwise, it checks whether any rewrite rule $\omega \in \Omega$ matches E, meaning that the left-hand side of ω can be unified with E. If so, it returns true, and false otherwise.

$$\textsc{Base}\ \frac{\mathsf{Root}(P) = v \qquad P \vdash v : (l, \phi) \qquad l \in \{\Box, \hat{\mathcal{I}}\}}{\Omega \vdash P \hookrightarrow \bot}$$

$$\textsc{Rec}\ \frac{\begin{array}{c}\mathsf{Root}(P) = v \qquad P \vdash v : (\mathsf{f}, \phi) \\ \in \mathsf{Children}(P, v) = \{v_1, \dots, v_n\} \qquad \Omega \vdash \mathsf{Subtree}(P, v_i) \hookrightarrow b_i\end{array}}{\Omega \vdash P \hookrightarrow (\exists_i.\ b_i = \top \vee \exists \omega \in \Omega.\mathsf{Is}(P, \omega))\ ?\ \top : \bot}$$

Fig. 14. Inference rules for Reducible. Ω represents all rewrite rules, some of which are shown in Figure 13.

Example 5.11. Consider a partial program of the form $\mathsf{Union}(P_1, P_2, \Box)$ where P_1, P_2 have been partially evaluated as $\hat{\mathcal{I}}_1$ and $\hat{\mathcal{I}}_2$, respectively. Suppose that the symbolic image $\hat{\mathcal{I}}_1$ is the set of objects $\{o_1, o_2, o_3\}$ and $\hat{\mathcal{I}}_2$ is $\{o_2, o_3\}$. Since $\hat{\mathcal{I}}_2 \subseteq \hat{\mathcal{I}}_1$, this program will match with the rewrite rule

$$\mathsf{Union}(A_1, \dots, A_i, \dots, A_n) \rightsquigarrow \mathsf{Union}(A_1, \dots, A_n) \text{ if } \exists j \text{ such that } A_i \subseteq A_j.$$

which corresponds to the domination rule for sets. Thus, this program simplifies to $\mathsf{Union}(P_1, \Box)$, meaning that the Reducible procedure will return $\top$ and this partial program will be pruned.

6 IMPLEMENTATION

We have implemented the proposed algorithm as a new tool called ImageEye written in Python. In what follows, we describe key implementation details that are not covered in the technical sections.

Computer vision primitives. Recall that our DSL operates over symbolic images, which are generated from the raw input image by applying existing computer vision primitives. In our implementation, we use the Amazon Rekognition library for object classification, text detection, and facial attribute classification. Compared with similar vision libraries, Rekognition offers more capabilities that are well-suited for image manipulation tasks of interest to this work.

Graphical user interface. ImageEye also incorporates a graphical user interface that allows users to demonstrate the desired image processing task. Our GUI is implemented in JavaScript and supports both image manipulation as well as image search. To use the GUI, the user first uploads their batch of images and then selects one or more images to annotate. For each image being annotated, the GUI indicates regions of the image that are classified as an object with rectangular bounding boxes. In the image editing mode, the user can select one of these objects and then apply the desired action (e.g., crop, blur, or highlight). When using ImageEye in search mode, the user can indicate the image as either being of interest or irrelevant. Once the user is done annotating a representative set of images, they press a button to invoke the synthesizer. If synthesis is successful, ImageEye applies the generated program to the entire image set and uploads the output to a new directory, which contains all the relevant images with the desired edits applied to them.

7 EVALUATION

In this section, we describe the results of our experimental evaluation, which is designed to answer the following research questions:

- **RQ1.** Can ImageEye automate interesting image manipulation and exploration tasks?
- **RQ2.** How many examples does ImageEye need to synthesize the intended program?
- **RQ3.** How does ImageEye's synthesis algorithm compare against existing baselines?
- **RQ4.** How important are the pruning techniques used by the synthesizer?
- **RQ5.** How effective are the synthesized programs in producing the desired edit on the test set?

Benchmarks. To answer these questions, we collected a set of 50 benchmark tasks across three domains, namely Wedding, Receipts, and Objects. Tasks in the Wedding domain involve identifying

Table 1. Statistics about images and tasks for each domain. Program size is measured in terms of AST nodes.

Dataset	# Images	Avg. # Objects	# Tasks	Avg. Program Size
Wedding	121	10	16	9.4
Receipts	38	59	13	7.8
Objects	608	3	21	8.3

and manipulating specific faces. An example task in this domain is to "crop out wedding guests who are not smiling." Tasks in the Receipts domain involve identifying specific words or classes of text, such as "highlight the prices to the right of the words 'total' and 'subtotal.'" Tasks in the Objects domain require manipulating specific classes of objects that are spatially related to other objects, such as, "crop the faces of people playing the guitar". Many of these tasks are motivated by real-world scenarios found on image editing forums, such as Reddit groups related to Photoshop and GIMP. For each task, we manually wrote a ground truth program in our DSL that can be used to check the correctness of the program returned by ImageEye.

Table 1 gives some statistics about each of the three domains used in our evaluation. As we can see, each domain varies in terms of the number of images they contain and the average number of objects in a given image. Observe that the Receipts domain contains the largest number of objects per image because each word is identified as a unique text object. In contrast, images in the Objects domain are much more sparse. For each domain, we have between 13 and 21 synthesis tasks, and the average size (in terms of AST nodes) of the ground truth program is in the 8-10 range.

7.1 Experimental Setup

To answer our first research question, we attempted to use ImageEye to automate each of our 50 benchmark tasks using the following methodology: We first select an image from the task's domain and apply the desired edit. When choosing an image, we prefer those that contain as few objects as possible, as this choice involves the least amount of work for the user. Then, we use ImageEye to synthesize a program based on this single demonstration. If the generated program produces the desired edit on *all* images in the data set, we consider the task to be successfully automated. Otherwise, we select a single image where ImageEye does *not* produce the desired edit and re-attempt synthesis with this *additional* example. We continue this process for up to 10 rounds and up to 180 seconds per round. All of our experiments are conducted on a desktop machine with 2.3 GHz dual-core Intel core i5 CPU and 8 GB of physical memory.

7.2 Main Results

Table 2 presents the results of this experiment. The key takeaway is that ImageEye can successfully automate 48 of the 50 tasks in our benchmark suite within the given resource limits. Table 2 also shows average and median synthesis times for the last round of user interaction. As we can see from this table, average synthesis time is around 15 seconds, with the median being much faster at around 1 second. We also note that synthesis time varies significantly across the domains, with the fastest being Objects and slowest being Receipts. This discrepancy makes sense considering the average number of objects per domain. In particular, recall that the number of constants in the DSL depends on the number of objects in the target domain, so synthesis generally takes longer in domains like Receipts that contain a lot of objects. However, the Receipts domain generally requires fewer rounds of user interaction, as object-dense images are richer in information. The last column of Table 2 shows the average number of rounds of user interaction. As we can see, the average number of demonstrations required across all three domains is just below 4.

Table 2. Summary of results for ImageEye. We include 95% confidence intervals.

Dataset	# solved	Avg. Synth Time (s)	Med. Synth Time (s)	Avg. # Examples
Wedding	14/16	15.6 ± 13.4	5.5	5.4 ± 1.0
Receipts	13/13	25.4 ± 23.4	1.6	2.2 ± 0.65
Objects	21/21	3.2 ± 2.4	0.1	3.8 ± 0.5
Total	**48/50**	**12.8 ± 8.0**	**1.2**	**3.8 ± 0.5**

Failure analysis. We now examine the two tasks that ImageEye fails to successfully automate. One of these tasks is from the Wedding domain and requires cropping the image to feature just the bride and the people standing directly to her left and right. In this case, ImageEye fails to find the correct program within the time limit of 180 seconds because the size of the ground truth program is fairly large and there are a large number of detected objects. The second task that ImageEye fails to automate is also in the Wedding domain and involves identifying images that contain the bride's face only when there are people standing directly to her left and right. For this benchmark, ImageEye requires more than 10 rounds of user interaction to find the desired program. Since this task requires extracting the bride's face only in a specific circumstance, there are many simpler programs that produce the same output on nearly all photos in the dataset.

Result for RQ1: ImageEye automates 48 out of 50 interesting image manipulation and exploration tasks, with a median synthesis time of 1.1 seconds.

Result for RQ2: ImageEye requires an average of 4 images to synthesize the intended program.

7.3 Comparison with Other Synthesis Tools

To answer our third research question, we compare the synthesis engine of ImageEye with existing synthesis tools. However, since existing tools do not support the image editing domain, we first reduce our learning problem to PBE (as discussed in Sections 4 and 5.2). Furthermore, since prior work does not consider DSLs that operate over images, we cast our synthesis problem as an instance of *syntax-guided synthesis* (SyGuS) and instantiate the SyGuS framework with our domain-specific language. Among the solvers that support the SyGuS format, we compare ImageEye's synthesis engine against the two most recent winners of the SyGuS competition. One of these solvers [Barbosa et al. 2022] extends the CVC SMT solver [Barrett et al. 2011] to support syntax-guided synthesis. The second one, EUSolver [Alur et al. 2017], is based on bottom-up enumerative search with equivalence reduction and uses a divide-and-conquer approach to decompose the synthesis task into smaller problems.

Among these solvers, we found the CVC solver to be ineffective at solving the synthesis problems that arise in our setting. In particular, instantiating our DSL in the CVC framework requires using the theory of sets (to represent symbolic images), but the resulting synthesis problems in this background theory are not easily solvable using a purely theorem proving approach. In fact, we found that this SMT-based approach is unable to solve even the simplest of our synthesis tasks within the given time limit.

In contrast, we were able to successfully instantiate EUSolver to handle the synthesis tasks from our image editing domain. The results of the comparison against EUSolver are presented in Figure 15 as a bar graph. Here, the x-axis indicates the difficulty level of the synthesis tasks (as measured by AST size); thus, bars in this plot correspond to synthesis tasks of increasing difficulty. On the other hand, the y-axis shows the number of tasks completed within the given time limit. The solid blue bars correspond to the results for ImageEye, and the hatched orange bars correspond

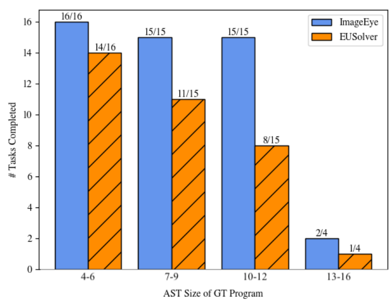

Fig. 15. Comparison of ImageEye and EUSolver.

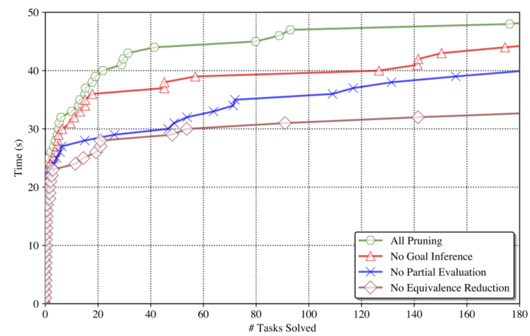

Fig. 16. Ablation study for ImageEye

to those of EUSolver. As we can see from this figure, EUSolver can solve 14 out of 16 of the easiest tasks, but, as the difficulty level increases, there is a growing gap between ImageEye and EUSolver. Overall, ImageEye can solve 14 more tasks than EUSolver out of the 50 tasks total.

To gain some intuition about these results, we briefly discuss why ImageEye outperforms EUSolver on our benchmarks. First, unlike EUSolver which is a generic solver, ImageEye performs a form of abstract interpretation customized to images and our image editing DSL. This type of reasoning allows ImageEye to prune many infeasible programs that need to be enumerated by EUSolver. Second, many of the techniques in EUSolver target branching, but our DSL allows branching in a stylized manner (at the top level and as part of filtering constructs). Finally, EUSolver works by combining sub-programs that work on a subset of examples, and this particular decomposition strategy does not seem effective in the image domain. For these reasons, ImageEye is more effective at solving the PBE problems that arise in the context of image extractor synthesis.

Result for RQ3: The baseline synthesis tool, EUSolver, can successfully solve 68% of the benchmarks compared with 96% solved by ImageEye.

7.4 Ablation Study

To answer our final research question, we present the results of an ablation study in which we disable some of the key components of our synthesis algorithm. In particular, we consider the following three ablations of ImageEye:

- **No Goal Inference:** This ablation does not use the goal inference technique of Section 5.3. However, it does perform equivalence reduction with partial evaluation and term rewriting.
- **No Partial Evaluation:** This version of ImageEye does not perform partial evaluation before applying the term rewrite rules from Section 5.5. However, it does perform goal inference and uses rewrite rules to prune the search space.
- **No Equivalence Reduction:** This ablation does not perform equivalence reduction using term rewriting. In other words, it does not utilize the techniques described in Section 5.5.

The results of this ablation study are presented as a cactus plot in Figure 16. Here, the x-axis shows cumulative synthesis time and the y-axis shows the number of benchmarks solved within a given time. As we can see from this figure, all of our proposed techniques have a significant impact on synthesis time. Without goal inference, ImageEye times out on four additional tasks and takes around 14 seconds longer on average to solve the tasks on which it does not time out. Without partial evaluation, ImageEye times out on eight additional tasks and takes around 23 seconds longer on average. Finally, without equivalence reduction, ImageEye times out on 16 additional tasks.

Result for RQ4: The techniques discussed in Sections 5.3-5.5 are important for making synthesis effective in the image editing domain.

7.5 Reliability of Underlying Neural Models

When reporting our main experimental results in Section 7.2, we manually inspect the synthesized program and consider the synthesis result to be correct if it is semantically equivalent to the ground truth program we wrote by hand. However, because the synthesized programs contain neural networks for object recognition and classification, even a correct synthesized program may not produce the expected output for all images in the test set. For instance, if the desired edit is to blur all cats in an image, and the object classification model does not recognize a specific cat in an image, then the program Is(Object(cat)) will not produce the desired output.

In this section, we additionally evaluate the accuracy of the synthesized programs in terms of the percentage of images in the test set for which the desired output is produced. However, since there are a very large number of images in some of the data sets, we randomly sample 20 images from each of the three data sets[2] and manually examine if the synthesized program produces the intended output for each of these 20 images.

Overall, across the three domains, we find that the synthesized programs produce the intended output on **87%** of the sampled images. Many of the failure cases stem from the same misclassification occurring numerous times. For instance, for the wedding data set, the face recognition model fails to identify that a specific wedding guest is smiling across many images.

Result for RQ5: The programs synthesized by ImageEye produce the desired edit for 87% of the images in the test set.

8 LIMITATIONS

In this section, we discuss some of the main limitations of ImageEye. First, the effectiveness of ImageEye is highly dependent on the underlying neural components. For example, if the target task involves a class of objects that the model cannot reliably identify, ImageEye will not be effective in producing the intended output image. However, we try to mitigate this problem through the choice of the neural primitives included in the DSL and intentionally exclude object classifiers that do not work reliably in practice. Additionally, we note that manually refining a small portion of the images in the data set is preferable over editing all images manually.

A second limitation of ImageEye is due to its user interaction model. In particular, to decide whether the synthesized program is correct, the user needs to inspect all images in the dataset, and, even then, it may be hard to distinguish whether any problems in the output are due to the lack of sufficient demonstrations or due to limitations of the neural primitives in the synthesized program. Additionally, if the underlying neural primitives misclassify relevant objects in the *training* example, then the user will not be able to perform their demonstration, as the ImageEye GUI only allows editing objects recognized by the object recognition engine. This design choice is intentional in that the interface *forces* the user to perform demonstration on “good" images. However, a potential disadvantage is that the user may need to go through multiple images before they find one on which the demonstration can be performed. One way to address both of these limitations could be through active learning approaches that suggest images for the user to label.

Finally, ImageEye is limited by the expressivity of the DSL, which only provides built-in functions like GetAbove and GetRight that can be evaluated easily by using bounding boxes. These functions are not suitable for reasoning about three-dimensional spatial concepts like one object being

[2]To ensure relevance of the sampled images, we re-sample if the output of the synthesized program is empty on that image.

behind another. This limitation could be addressed by extending the DSL with functions that are implemented using additional neural primitives.

9 RELATED WORK

Image manipulation. Image manipulation is a long-standing problem in computer vision, graphics, and computational photography. Recent efforts in this space have used deep neural networks to generate realistic variants of a given image, applying them to tasks like inpainting [Nazeri et al. 2019; Xiong et al. 2019; Yu et al. 2019], extrapolation [Wang et al. 2019; Zhou et al. 2018], and photo editing [Brock et al. 2017; Choi et al. 2018; Lample et al. 2017; Zhao et al. 2018]. As a representative example of a work in this space, Fader [Lample et al. 2017] can generate variants of a subject with different attributes like age or gender. In this work, we solve a different type of image manipulation problem than most of these prior efforts: our focus is on identifying what operations to apply to *which parts* of the image, rather than generating realistic variants of a given input image. Furthermore, our approach is based on neurosymbolic program synthesis rather than generative neural networks. However, these approaches can be incorporated into our overall approach by treating them as pre-trained neural network primitives in our DSL.

Neurosymbolic programming for images. Recently, there has been growing interest in using neurosymbolic DSLs that include both logical and neural components in the image domain [Ellis et al. 2018; Huang et al. 2020; Johnson et al. 2017; Mao et al. 2019; Reed and de Freitas 2016; Tian et al. 2019; Young et al. 2019]. Similar to our work, these efforts typically combine symbolic operators for higher-level reasoning with neural modules for perception. The closest work in this space is that of Huang et al. [Huang et al. 2020], which generates programmatic *referring expressions* that identify specific objects in terms of their attributes and relationships with respect to other objects in the image. However, their work differs from ours in several respects: (1) They focus on locating a single object whereas we focus on applying actions to a *set* of objects; (2) they synthesize logic programs using a different synthesis algorithm based on deep Q-learning and hierarchical search; and (3) their focus is on a synthetic dataset with geometric shapes whereas our focus is on more realistic images with faces, text, and arbitrary objects.

Top-down enumerative search. Several recent synthesis techniques use a combination of top-down enumerative search and lightweight deductive reasoning to significantly reduce the search space [Albarghouthi et al. 2013; Feng et al. 2018; Feser et al. 2015; Lubin et al. 2020; Osera and Zdancewic 2015; Polozov and Gulwani 2015; Wang et al. 2017]. Among these, our approach bears similarities to enumerative synthesis approaches that propagate the goal to the missing subexpressions. In particular, Myth [Osera and Zdancewic 2015], SMyth [Lubin et al. 2020] and λ^2 [Feser et al. 2015] infer new input-output examples for the holes in a partial programs by utilizing type information embedded in the language. While our method also performs goal-directed reasoning, the underlying deductive reasoning techniques are different. Another synthesis framework that uses example-based specifications is FlashMeta [Polozov and Gulwani 2015], which propagates specifications from the DSL operators down into their arguments using so-called *witness functions*. Unlike the synthesis algorithm we present here, FlashMeta uses version space algebras (VSA) to represent the space of all programs that are consistent with the provided input-output examples.

Prior efforts on regular expression synthesis [Chen et al. 2020; Lee et al. 2016; Ye et al. 2021] also utilize over- and under-approximations to eliminate infeasible programs. In particular, Regel [Chen et al. 2020] and AlphaRegex [Lee et al. 2016] both derive over- and under-approximations of the set of strings that could be matched by a partial regex. In contrast, we use over- and under-approximations in a different context and approximate the synthesis sub-goals as opposed to the outputs of a given partial program.

Synthesis using term rewriting. There has been several efforts that use term rewriting [DERSHOWITZ and JOUANNAUD 1990] to speed up program synthesis [Dershowitz and Reddy 1993; Reddy 1989; Smith and Albarghouthi 2019; Yaghmazadeh et al. 2018]. These techniques have found applications in many domains, including CAD model construction [Nandi et al. 2020], robotic process automation [Dong et al. 2022], compiler construction [Visser et al. 1998], and writing numerical software [Boyle et al. 1997]. Similar to the work of Smith et al. [Smith and Albarghouthi 2019], we also use an equational rewrite system to reduce the number of partial programs enumerated during top-down synthesis; however, our technique combines this idea with partial evaluation [Jones et al. 1993] and goal-directed reasoning to make it more effective.

Synthesis using partial evaluation. There are a variety of domain-specific [Solar-Lezama 2008; Torlak and Bodik 2013] and domain-agnostic [Feng et al. 2017; Holtz et al. 2021] synthesis techniques that use partial evaluation [Jones et al. 1993] to obtain a more efficient synthesis procedure. In particular, Morpheus [Feng et al. 2017] utilizes partial evaluation to infer a more precise specification of the partial program, which helps to increase its SMT-based pruning power. Similar to our approach, both Rosette [Torlak and Bodik 2013] and IDIPS [Holtz et al. 2021] evaluate the concrete part of the partial program to obtain a simplified version. However, our work differs from these prior techniques in that we use partial evaluation to make term rewriting more effective.

Programming by demonstration. Programming-by-demonstration techniques [Lau and Weld 1998] utilize user demonstrations to learn a new task. This paradigm has been successfully adopted in a variety of scenarios, including web automation [Barman et al. 2016; Chasins and Bodik 2017; Dong et al. 2022; Lin et al. 2009], robot learning [Argall et al. 2009; Billard et al. 2008; Dillmann and Friedrich 1996], text editing [Lau et al. 2003], and SQL query synthesis [Zhou et al. 2022]. ImageEye also allows users to demonstrate the desired task through a graphical user interface and leverages the demonstration to decompose the synthesis task into a set of PBE problems, one for each action in the demonstration.

10 CONCLUSION

We have presented a new synthesis-based approach for automating image editing and search tasks. Given a few user demonstrations performed through a graphical user interface, our method synthesizes a program that can be used to automate the desired image search or batch editing task. At the heart of our approach lies a neuro-symbolic DSL that combines functional operators with pre-trained neural modules for object detection and classification. We have implemented this approach in a new tool called ImageEye and evaluated it on 50 image search and editing tasks across three different domains involving human faces, text, and arbitrary objects. Our evaluation shows that ImageEye can automate 96% of these tasks, with a median synthesis time of 1 second and requiring on average four user demonstrations.

ACKNOWLEDGMENTS

We would like to thank Michelle Ding, fellow graduate students on GDC 5S, and the anonymous reviewers for their help and feedback on this paper. This material is based upon work supported by the National Science Foundation under grant numbers CCF-1811865 and CCF-1918889, Google under the Google Faculty Research Grant, as well as Facebook, Amazon, Intel, and RelationalAI.

DATA AVAILABILITY

An artifact supporting the results of this paper is available on Zenodo [Barnaby et al. 2023a]. It includes the ImageEye implementation, benchmarks, and benchmarking scripts.

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 934–950.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 10205. Springer Berlin Heidelberg, Berlin, Heidelberg, 319–336.

Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A survey of robot learning from demonstration. *Robotics and Autonomous Systems* 57, 5 (2009), 469–483. https://doi.org/10.1016/j.robot.2008.10.024

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: Web Automation by Demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 748–764. https://doi.org/10.1145/2983990.2984020

Celeste Barnaby, Qiaochu Chen, Roopsha Samanta, and Isil Dillig. 2023a. *ImageEye: Batch Image Editing with Program Synthesis*. https://doi.org/10.5281/zenodo.7810841

Celeste Barnaby, Qiaochu Chen, Roopsha Samanta, and Isil Dillig. 2023b. ImageEye: Batch Image Processing Using Program Synthesis. https://doi.org/10.48550/arXiv.2304.03253

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

Aude Billard, Sylvain Calinon, Rüdiger Dillmann, and Stefan Schaal. 2008. *Robot Programming by Demonstration*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1371–1394. https://doi.org/10.1007/978-3-540-30301-5_60

James M. Boyle, Terence J. Harmer, and Victor L. Winter. 1997. *The TAMPR Program Transformation System: Simplifying the Development of Numerical Software*. Birkhäuser Boston, Boston, MA, 353–372. https://doi.org/10.1007/978-1-4612-1986-6_17

Andrew Brock, Theodore Lim, J.M. Ritchie, and Nick Weston. 2017. Neural Photo Editing with Introspective Adversarial Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HkNKFiGex

Sarah Chasins and Rastislav Bodik. 2017. Skip Blocks: Reusing Execution History to Accelerate Web Scripts. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 51 (oct 2017), 28 pages. https://doi.org/10.1145/3133875

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

Yunjey Choi, Minje Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, and Jaegul Choo. 2018. StarGAN: Unified Generative Adversarial Networks for Multi-domain Image-to-Image Translation. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8789–8797. https://doi.org/10.1109/CVPR.2018.00916

Nachum DERSHOWITZ and Jean-Pierre JOUANNAUD. 1990. CHAPTER 6 - Rewrite Systems. In *Formal Models and Semantics*, JAN VAN LEEUWEN (Ed.). Elsevier, Amsterdam, 243–320. https://doi.org/10.1016/B978-0-444-88074-1.50011-1

Nachum Dershowitz and Uday S. Reddy. 1993. Deductive and inductive synthesis of equational programs. *Journal of Symbolic Computation* 15, 5 (1993), 467–494. https://doi.org/10.1016/S0747-7171(06)80002-7

R. Dillmann and H. Friedrich. 1996. Programming by demonstration: A machine learning approach to support skill acquision for robots. In *Artificial Intelligence and Symbolic Mathematical Computation*, Jacques Calmet, John A. Campbell, and Jochen Pfalzgraf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 87–108.

Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. 2022. WebRobot: Web Robotic Process Automation Using Interactive Programming-by-Demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing

Machinery, New York, NY, USA, 152–167. https://doi.org/10.1145/3519939.3523711

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/6788076842014c83cedadbe6b0ba0314-Paper.pdf

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. *SIGPLAN Not.* 53, 4 (jun 2018), 420–435. https://doi.org/10.1145/3296979.3192382

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Jarrett Holtz, Simon Andrews, Arjun Guha, and Joydeep Biswas. 2021. Iterative Program Synthesis for Adaptable Social Navigation. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Prague, Czech Republic). IEEE Press, 6256–6261. https://doi.org/10.1109/IROS51168.2021.9636540

Jiani Huang, Calvin Smith, Osbert Bastani, Rishabh Singh, Aws Albarghouthi, and Mayur Naik. 2020. Generating Programmatic Referring Expressions via Program Synthesis. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 4495–4506. https://proceedings.mlr.press/v119/huang20h.html

Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C. Lawrence Zitnick, and Ross Girshick. 2017. Inferring and Executing Programs for Visual Reasoning. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., USA.

Guillaume Lample, Neil Zeghidour, Nicolas Usunier, Antoine Bordes, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2017. Fader Networks: Manipulating Images by Sliding Attributes. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 5969–5978.

Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (2003), 111–156. https://doi.org/10.1023/A:1025671410623

Tessa A. Lau and Daniel S. Weld. 1998. Programming by Demonstration: An Inductive Learning Formulation. In *Proceedings of the 4th International Conference on Intelligent User Interfaces* (Los Angeles, California, USA) *(IUI '99)*. Association for Computing Machinery, New York, NY, USA, 145–152. https://doi.org/10.1145/291080.291104

Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 542–553. https://doi.org/10.1145/2594291.2594333

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Amsterdam, Netherlands) *(GPCE 2016)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/2993236.2993244

James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-User Programming of Mashups with Vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces* (Sanibel Island, Florida, USA) *(IUI '09)*. Association for Computing Machinery, New York, NY, USA, 97–106. https://doi.org/10.1145/1502650.1502667

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 109 (aug 2020), 29 pages. https://doi.org/10.1145/3408991

Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. 2019. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJgMlhRctm

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

Kamyar Nazeri, Eric Ng, Tony Joseph, Faisal Qureshi, and Mehran Ebrahimi. 2019. EdgeConnect: Structure Guided Image Inpainting using Edge Prediction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*

Workshops.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2737924.2738007

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 107–126. https://doi.org/10.1145/2814270.2814310

Uday S. Reddy. 1989. Rewriting techniques for program synthesis. In *Rewriting Techniques and Applications*, Nachum Dershowitz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–403.

Scott E. Reed and Nando de Freitas. 2016. Neural Programmer-Interpreters. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.06279

Calvin Smith and Aws Albarghouthi. 2019. Program Synthesis with Equivalence Reduction. In *Verification, Model Checking, and Abstract Interpretation*, Constantin Enea and Ruzica Piskac (Eds.). Springer International Publishing, Cham, 24–47.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rylNH20qFQ

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 13–26. https://doi.org/10.1145/289423.289425

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

Yi Wang, Xin Tao, Xiaoyong Shen, and Jiaya Jia. 2019. Wide-Context Semantic Image Extrapolation. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 1399–1408. https://doi.org/10.1109/CVPR.2019.00149

Wei Xiong, Jiahui Yu, Zhe Lin, Jimei Yang, Xin Lu, Connelly Barnes, and Jiebo Luo. 2019. Foreground-Aware Image Inpainting. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 5833–5841. https://doi.org/10.1109/CVPR.2019.00599

Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-Example. *Proc. VLDB Endow.* 11, 5 (oct 2018), 580–593. https://doi.org/10.1145/3187009.3177735

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2021. Optimal Neural Program Synthesis from Multimodal Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. Association for Computational Linguistics, Punta Cana, Dominican Republic, 1691–1704. https://doi.org/10.18653/v1/2021.findings-emnlp.146

Halley Young, Osbert Bastani, and Mayur Naik. 2019. Learning Neurosymbolic Generative Models via Program Synthesis. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 7144–7153. https://proceedings.mlr.press/v97/young19a.html

Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas Huang. 2019. Free-Form Image Inpainting With Gated Convolution. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 4470–4479. https://doi.org/10.1109/ICCV.2019.00457

Bo Zhao, Bo Chang, Zequn Jie, and Leonid Sigal. 2018. Modular Generative Adversarial Networks. In *Computer Vision – ECCV 2018*, Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss (Eds.). Springer International Publishing, Cham, 157–173.

Xiangyu Zhou, Rastislav Bodik, Alvin Cheung, and Chenglong Wang. 2022. Synthesizing Analytical SQL Queries from Computation Demonstration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 168–182. https://doi.org/10.1145/3519939.3523712

Yang Zhou, Zhen Zhu, Xiang Bai, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. 2018. Non-Stationary Texture Synthesis by Adversarial Expansion. *ACM Trans. Graph.* 37, 4, Article 49 (jul 2018), 13 pages. https://doi.org/10.1145/3197517.3201285

Received 2022-11-10; accepted 2023-03-31